Haaga-Helia
University of Applied Sciences

Arduino Management Interface using React

Loïc PROD’HOM

Bachelor’s Thesis
Degree Programme in BITE
2020

Haaga-Helia
University of Applied Sciences

Abstract

12/03/2020

Author(s) Loïc PROD'HOM	
Degree programme BITE	
Report/thesis title Arduino Management Interface using React	**Number of pages and appendix pages** **41 + 1**

The objective of this thesis is to develop a web interface to manage and control Arduino devices. This project is commissioned by my former employer, the Geneva School of Business Administration, and aims to deliver a user-friendly interface that could potentially be used in lectures.

The thesis is structured into two main parts:

The first part is largely theoretical and lays the groundwork to understanding the key concepts of this project, as well as the technology used for communication: the MQTT protocol. A subchapter is dedicated to describing the MQTT protocol in detail, as well as its key components: the broker, the publisher and the subscriber. The last subchapter of that section explains why exploration of an Arduino-hosted RESTful architecture was ultimately discontinued for this project in favour of the lightweight MQTT protocol.

The second part follows the practical implementation of the project, beginning with the creation of the data structure on the Arduinos and the early mock-ups of the user interface. It then details the implementation of the Mosca broker, followed by implementation of a JavaScript publisher for testing purposes, as well as implementation of an Express API to link front-end to the broker, and finally of the implementation of the React interface itself.

This application is complete, and its source code can be found on the provided GitHub repositories (see Appendix 1). This thesis document has been written in a didactic style to help anyone looking for a way to manage such devices for their own projects, or simply looking to implement MQTT communication for their React applications in an environment with limited bandwidth.

Keywords
Arduino, React, UI, MQTT, Webserver, JavaScript

Table of contents

Table of Figures

Technical Terms and abbreviations used

Ampere: Unit of electrical current measurement (will mostly be noted as "A").(RapidTables, 2020)

Central Processing Unit: Electronic component that executes the majority of an electronic device's programmed instructions (will be referenced as "CPU").(Vangie Beal, 2020)

Random Access Memory: Also known as RAM, it is the main memory used for software execution and working data storage. (Vangie Beal, 2020)

JSON: "JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999."(JSON, 2020)

MAC Address: A Media Access Control address is a device's unique physical address.(Mahesh Parahar, 2019). It is the address used to communicate on the local network.

IP Address: An Internet Protocol Address is a device's logical address, unique on the local network. (Mahesh Parahar, 2019). It is used to communicate with devices on other remote networks.

DHCP: The Dynamic Host Configuration Protocol is a protocol used to assign an IP address to a device dynamically. (Vangie Beal, 2020).Attribution of addresses is done by a DHCP Server.

API: An Application Programming Interface is "a software intermediary that allows two applications to talk to each other" (Shana Pearlman, 2016). They usually serve as the intermediary between two different applications with differing processes and software thanks to a protocol or methods known by both parties. (Shana Pearlman, 2016)

1 Introduction

1.1 Topic

Arduino are microcontrollers that can perform basic tasks or collect raw data such as temperature. However, over my years on developing on Arduinos, a glaring weakness I have noticed is the lack of a proper platform to handle and manage several devices at once on a network. Since Arduino is a fully open-source software development platform, tutorials are found on a plethora of subjects, but they often rely on using the Arduino as a webserver serving an HTML webpage for a simple end user interface (Arduino, 2020). This heavily restricts the other physical capabilities of the microcontroller, since a large portion of resources (primarily CPU and RAM usage) must be allocated to running the webserver instance and holding the webpage in memory. That is why another possibility is to use MQTT, a lightweight communication protocol, thanks to existing Arduino libraries supporting it. The downside is that using this protocol requires some advanced configuration on the server app. The goal here is to combine the resource efficiency of MQTT while retaining the user-friendliness of a web interface by outsourcing display to a React app running on an Express server. We should end up with a structure like the following diagram (Figure 1):

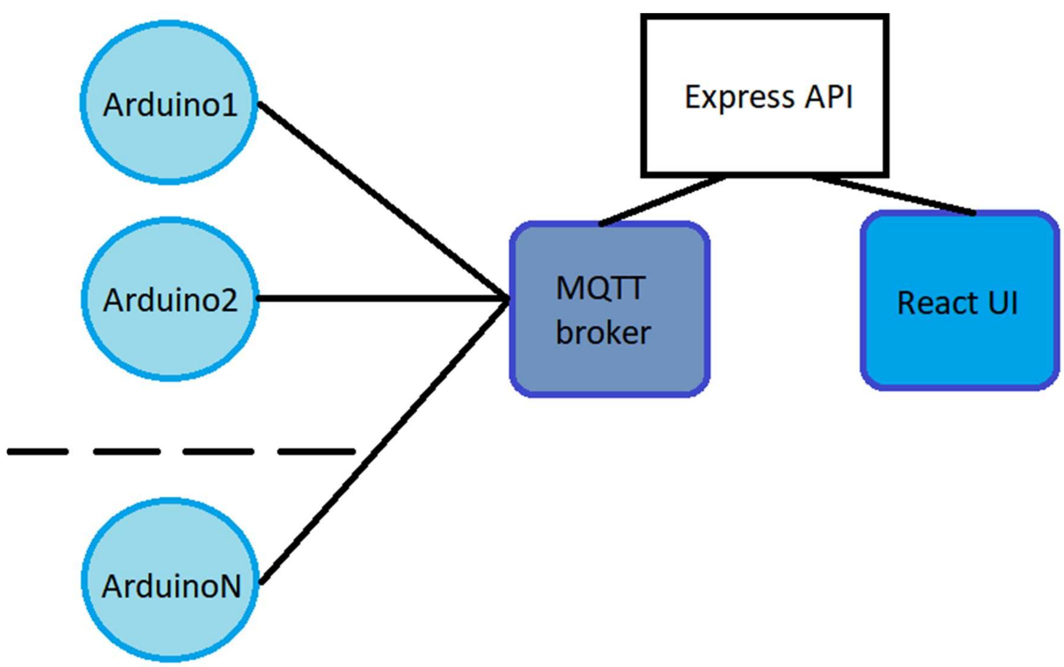

Figure 1 – General architecture of project components

1.2 Commissioning party

The commissioner for this project is my former employer, the Geneva School of Business Administration (mostly known as “Haute Ecole de Gestion Genève” or “HEG Genève”). I worked for the Internet network lecturers as an intern from October 2016 to September 2017. One of my tasks was to develop Arduino prototypes that would be mounted in boxes to accomplish various tasks while being controlled and monitored through HTTP Get requests. Since memory management was a constant struggle, this project should open more opportunities in terms of what the microcontrollers will be able to do in terms of external interactions.

2 Arduino microcontrollers and data communication

2.1 Introduction to Arduinos

An Arduino is a microcontroller designed for electronic development. In more concrete terms, it is a device akin to a tiny computer with minimalistic capabilities that can read data from input pins and emit electrical current from output pins. The standard Arduino Uno board has 13 pins that can emit or receive a digital signal, and 6 that can read analog signals. The board operates on 5V voltage and has a RAM of 2kB with a processor running at 16MHz (Arduino, 2020). To give a bit of perspective, the average PC has 8Gb of RAM with a processor running above a 2GHz frequency. This further emphasizes the need to preserve memory usage for Arduino applications.

Arduinos are generally used for basic applications, such as sending current to an LED to emit light, or reading voltage from sensors or buttons connected to the Arduino’s pins to parse data (calculations and formulas to convert the voltage into data are provided by the sensor’s manufacturer).

The Arduino executes a simple software usually stored in a single .ino file (referred to as a “sketch”), written in C and compiled into assembly by Arduino’s IDE before being uploaded to the board (Arduino, 2020). The code is divided into two essential procedures that must be declared for the Arduino to function: “*void setup()*” and “*void loop()*”. The first procedure is run once when the device is powered and serves initialisation purposes. The second one is looped until the device is powered off and is used to handle all events and actions during its runtime.

2.2 Arduino and Ethernet

Arduinos are an easy way to wire an array of sensors and collect data. However, the default Serial terminal on the Arduino IDE is quite limited in terms of display: it only prints out strings and allows sending strings or characters to the board. Serial communication in general is quite limited as it only allows a wired device to device connection (through a USB port in the case of Arduino-to-PC communication). This makes it difficult to deploy the microcontroller away from a computer. This is where the Arduino Ethernet library comes into play.
It is possible to connect an Ethernet board or device to the Arduino thanks to a built-in library called “SPI.h”. This library allows for Serial Peripheral Interface communication between the Arduino and other devices. Let us talk more in detail about that system to understand how they communicate: the Arduino is the “master” and can control one or several “slave” devices.

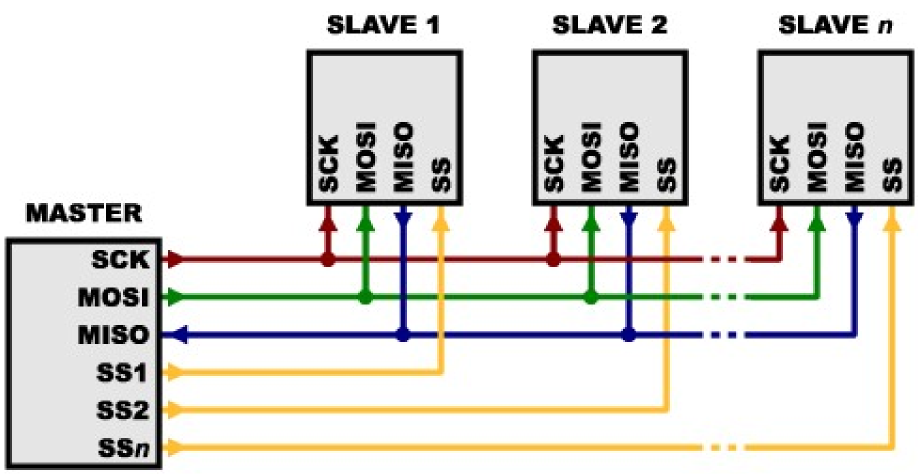

Figure 2 - Multiple Slaves (MikeGrusin, Sparkfun, 2020)

The master outputs data on its MOSI pin (Master Out, Slave In) and receives data on its MISO pin (Master In, Slave Out). The rate at which data is transferred is controlled by the master’s clock, sent to the slave devices through the SCK pin. In the case where several slave devices are present the Slave Select (SS) pin is used to inform the desired device that communication has begun. (MikeGrusin, 2013).

The Arduino Ethernet library controls an Ethernet device using this principle and the corresponding 4 pins. It requires basic network configuration to be able to send and receive data: the board’s MAC address, its IP address, the gateway’s IP address, and the subnet’s mask. It is possible to omit some of these parameters when initialising the Ethernet instance if a DHCP server is present on the network to allocate IP and gateway information automatically.

```
#include <Ethernet.h>
#include <SPI.h>


byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 10, 0, 0, 177 };
byte server[] = { 64, 233, 187, 99 }; // Google

EthernetClient client;

void setup()
{
  Ethernet.begin(mac, ip);
  Serial.begin(9600);

  delay(1000);

  Serial.println("connecting...");

  if (client.connect(server, 80)) {
    Serial.println("connected");
    client.println("GET /search?q=arduino HTTP/1.0");
    client.println();
  } else {
    Serial.println("connection failed");
  }
}

void loop()
{
  if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }

  if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
    for(;;)
      ;
  }
}
```

Figure 3 – Ethernet Client Example (Arduino, Arduino, 2020)

Above (Figure 3) is an example sketch to create a simple Web client that connects to Google's search engine and prints the results of a search on the serial terminal. Since the MAC and IP addresses are provided, the gateway and subnet mask will be requested to a DHCP server.

However, the limitations of this library are shown in the way that it simply prints out raw data once a connection has been established. Thus, to make a GET request, it must manually be written in the code. Error management is not automatic either, the developer has to manually verify that the connection has been established before printing out the data stream. Closing the connection is also manually handled according to application logic.

2.3 Arduino and MQTT

A viable alternative to HTTP communication with Arduinos is MQTT, or Message Queuing Telemetry Transport. It is a network protocol designed for networks with limited bandwidth or devices with few physical resources. The model for MQTT communication is quite simple: clients can publish on topics or subscribe to them, and a broker centralizes all the information for the clients.

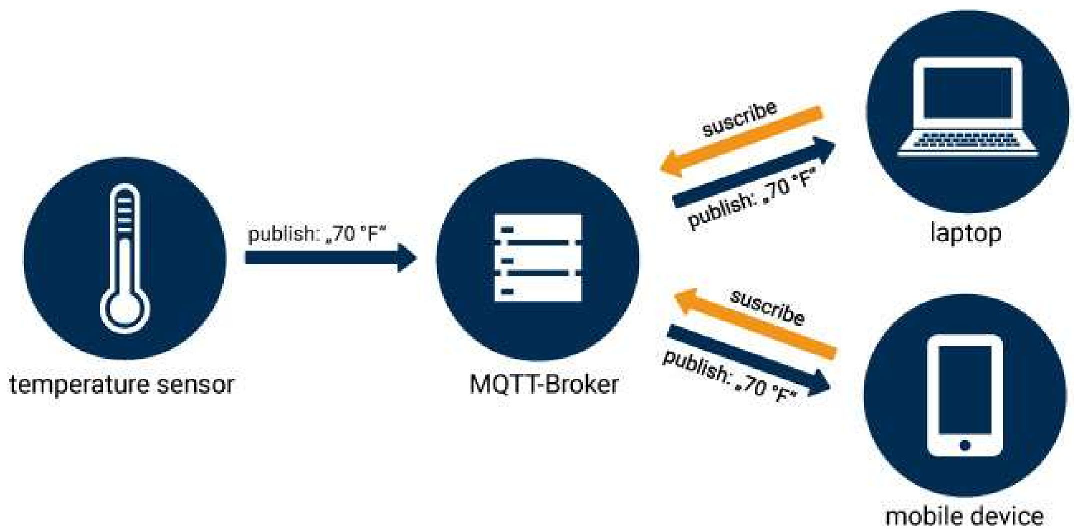

Figure 4 – What is an MQTT broker? (OPC, OPC Router, 2020)

Let us break down what this all means: the broker is akin to a server in this model, and data is structured through “topics”. Topics are categories that the broker uses to sort out incoming messages and deliver them to interested parties. In the above figure (Figure 4), a laptop and a mobile phone both indicate they want to receive information relative to the “temperature” topic with a “subscribe” message sent to the broker. When the sensor emits data to the broker through a “publish” message on the “temperature” topic, that information is relayed by the broker to all subscribed client with “publish” messages. (OPC, 2020)

In the context of this project, the idea is to use the Arduinos as publishing clients, a JavaScript Node server as the broker, and a React interface as a subscribing client. This means that our architecture will resemble the following figure (Figure 5):

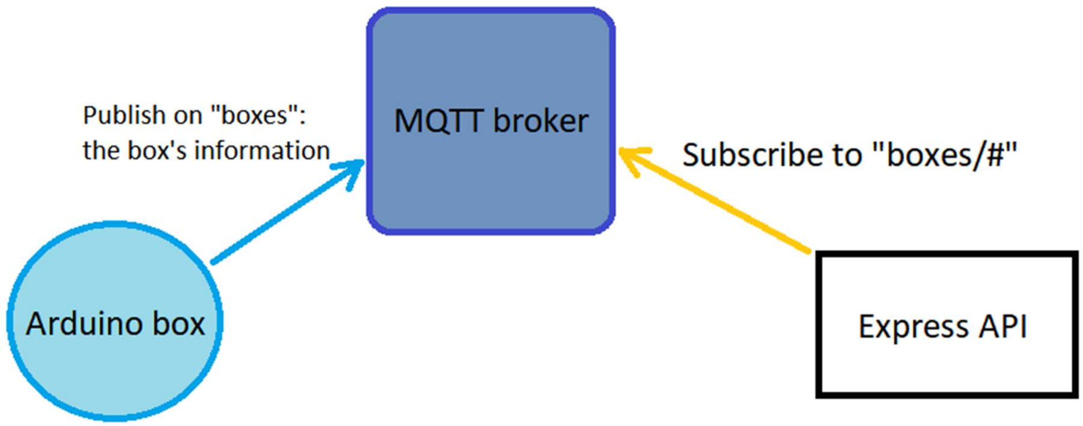

Figure 5 – MQTT Architecture part one

The Express API subscribes to the “boxes/#” topic on the broker to receive information on any box publishing any information (the “#” character here is a wildcard meaning “any topic and its subtopics”). Then, whenever a box comes online on the network, they publish under the “boxes” topic their information and subscribe to the topic related to them. The broker can then relay the publish message to the subscribed API so that it can be added to the list of known boxes. Then, whenever a user wishes to issue a command to the box through the UI, the React app sends a PUT request to the API with the box’s updated state, which is then processed by the API by editing its list of boxes and publishing the changes to the broker. That message is then relayed to the subscribed Arduino. Let us break down that process with an example to better understand how these components are meant to communicate.

We begin with the requirements of the MQTT broker and the Express API both being online, with the API app being subscribed to the “boxes/#” topic. Then an Arduino1, a relay type box, is plugged into the network and publishes to the “boxes” topic their information in a JSON payload like so (Figure 6):

```
state = {
  id: 1,
  IP: "192.168.0.91",
  mac: "dead.aaaa.feed",
  type: "Relay",
  data: {
    relay1Status: "ON",
    relay2Status: "ON",
    relay1: "3.5A",
    relay2: "2.5A",
  },
};
```

Figure 6 – Example Box State

The Arduino then immediately subscribes to the topic related to their ID to be notified when controls are issued.

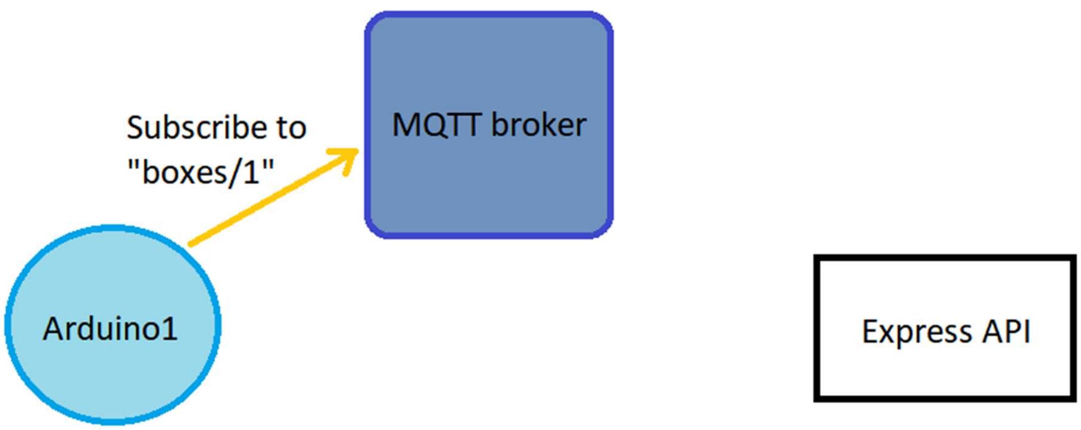

Figure 7 – MQTT Architecture part two

The topic tree is explained in more detail in section 4.1 of the thesis. After these initial messages, Arduino1 can then publish the data it collects during its runtime to the broker:

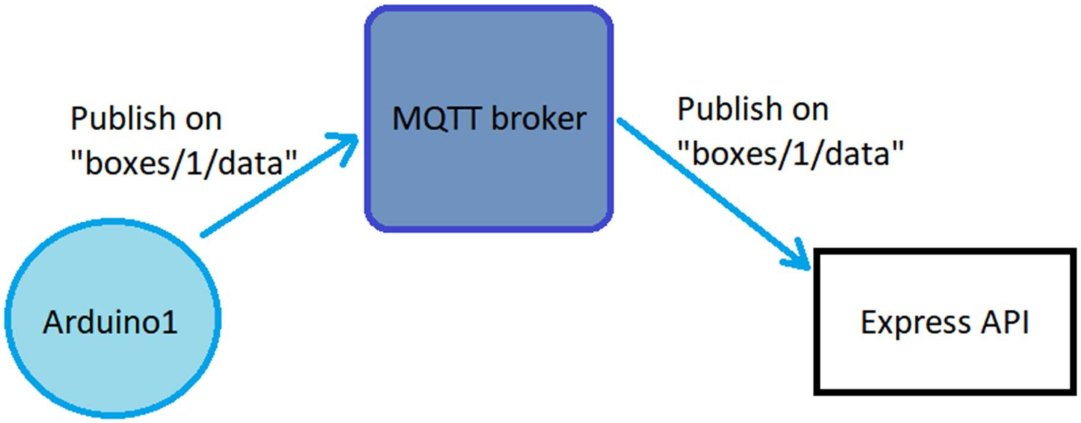

Figure 8 – MQTT Architecture part three

The Express API receives the data after every publish from the broker. At some point, a user wishes to close relay1 to switch off the PC that is plugged into it (the PC itself is not represented on the diagram (Figure 9) since it does not interact with our MQTT conversation in any way). A publish message is sent on the “boxes/1” topic:

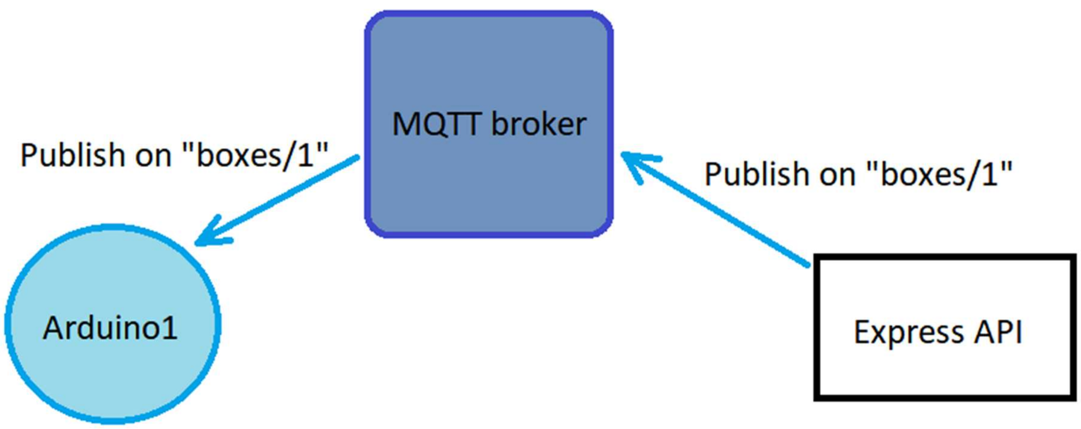

Figure 9 – MQTT Architecture part four

The publish message issued in this visual (Figure 9) contains the new state of the Arduino in its payload as a JSON object like so (Figure 10):

```
newState = {
  id: 1,
  IP: "192.168.0.91",
  mac: "dead.aaaa.feed",
  type: "Relay",
  data: {
    relay1Status: "OFF",
    relay2Status: "ON",
    relay1: "3.5A",
    relay2: "2.5A",
  },
};
```

Figure 10 – Example New state

The Arduino then updates its internal state to the one it has just received, thus changing the state of relay1 to “OFF”.

For the Arduino client, Nick O’Leary’s MQTT Client for Arduino library should fit our needs, and for the broker, Mosca is an adequate choice to run on Node.js. Finally, for the sub-scribed Express API, we will be using Matteo Collina’s mqtt node package.

2.4 Why a React UI?

```
    if (request.indexOf('\r') != -1) {
      client.println("HTTP/1.1 200 OK");
      client.println("Content-Type: text/html");
      client.println("Connection: close");
      client.println();
      client.println("<!DOCTYPE HTML>");
      client.println("<html>");

//---partie envoi des donnees---------

      client.println("Serveur commandant 2 relais avec detection de tension et mesure de courant<br>");
      client.println("Le bouton \"Reset Mesures\" doit etre utilise apres ouverture de l'un des relais.<br>");
      client.println("(une fois que le pic de consommation passe)<br>");
      if (tension())
      {
        client.println("Tension de 240V dans la boite");
      }
      else {
        client.println("Pas de tension.");
      }
      client.println("<br>");
      client.println("Intensite sur A1: ");
      client.println(current1,4);
      client.println(" A. <br>");
      client.println("Intensite sur A2: ");
      client.println(current2,4);
      client.println(" A. <br>");

    //--Boutons de commande des relais--//
      client.println("<br><input type=\"button\" name=\"b1\" value=\"Relais 1 ON  \" onclick=\"location.href='/relay1ON'\">");
      client.println("<br>");
      client.println("<br><input type=\"button\" name=\"b2\" value=\"Relais 1 OFF\" onclick=\"location.href='/relay1OFF'\">");
      client.println("<br>");

      client.println("<br><input type=\"button\" name=\"b3\" value=\"Relais 2 ON  \" onclick=\"location.href='/relay2ON'\">");
      client.println("<br>");
      client.println("<br><input type=\"button\" name=\"b4\" value=\"Relais 2 OFF\" onclick=\"location.href='/relay2OFF'\">");
      client.println("<br><br><br>");

    //-------Reset des mesures----------//
      client.println("<br><input type=\"button\" name=\"b5\" value=\"Reset Mesures\" onclick=\"location.href='/reset'\">");
      client.println("<br>");
```

Figure 11 – Arduino Webserver snippet

The above figure (Figure 11) is a code snippet that prints out the webpage to clients requesting to interact with one of the boxes made at HEG. This particular box is capable of being plugged as a medium between electronic appliances and an electrical outlet, can measure the intensity of the current being drawn, and can switch the appliances off using electrical relays (A relay is similar to a switch in the sense that it either allows current to flow through a circuit or blocks it, the main difference being that its state is not changed by hand but by applying electrical current). The displayed interface shows some informational text with the electrical current values measured by the box, as well as buttons to interact with the relays, and a button to reset measures.

This approach is extremely limited in terms of user interface design since the HTML display must be printed line by line to the client. A more elaborate interface is not something that can easily be accomplished because of the Arduino’s biggest weakness: memory space.

It is indeed difficult to insert more intricate rendering logic such as JavaScript code to manipulate elements since the page is not only being printed line by line, but those lines are

String objects held in the Arduino's RAM during runtime (a RAM whose capacity does not exceed 2kB on the Arduino Uno model). That is why the favoured approach during development of these boxes was to minimize usage of these Strings as much as possible.
However, by outsourcing display and user interface to a React application, we can forego them entirely and strictly focus on data collection and event listening, which should allow the box to be more responsive thanks to not having the burden of running a server and listening to incoming connections.
An additional shortcoming of the "Arduino as webserver" method was that the webserver's IP address was attributed through DHCP, and thus could not be known unless the DHCP server's logs were checked. The workaround was to send a syslog message to a syslog server as soon as the box got an IP address, which requires extra configuration on the Arduino (and extra load on the processor to handle the Syslog client instance). With the MQTT model however, the box can publish its IP under a topic that the React interface is subscribed to. Streamlining technologies used helps maintain consistency and further reduces the number of libraries required for the Arduino to function.
An additional advantage of centralizing each interface to control each type of box on a single application is that maintenance and version controlling of the visual interface will be easier, as currently to install a new version of the sketch the Arduino must be unmounted from the box and updated manually through a USB connection, so if it is possible to limit that to critical changes in the core functionalities of the box, it will not be necessary to go and retrieve the box from wherever it is mounted quite as often.

2.5 MQTT vs Arduino-hosted REST

An alternative approach to using MQTT would be to rely on hosting a REST service for communication: a single JSON object on the Arduino would store its "state", such as its IP address, its box type, and information related to its type like the electrical relays' state in the case of the measuring boxes.
In theory, the Arduino would simply hold relevant values in its JSON state and adjust its electronic components' state accordingly. The Express API would then need to update specific values in the state and post it to the Arduino, and that is precisely where the main difference between the two approaches lies: the Arduino would still need to be used as a server to host a REST API for the interface to be able to post and request data. One of the goals for this project was to ease the burden on the Arduino's processor by removing the need for such a server instance in the memory. And so, the REST approach is less attractive than the MQTT approach.

3 Implementation of the project

This project will be divided into three core sections: definition of the data model and creation of mock-ups of the end interface, then development of an Arduino sketch implementing MQTT communication, and finally development of the React interface and MQTT broker.

The first part aims to lay the groundwork to the applicative logic of this project, first by structuring how data is saved and handled on the Arduinos and the interface, but also which React components should be used and how they should be arranged on the end user screen.
The second part will involve the implementation of the aforementioned data model into an Arduino sketch, as well as the functions and procedures necessary to MQTT communication.
The final part will be the implementation of the React components on the interface, as well as implementation of the Node.js-based MQTT broker and Express API.

3.1 Data models

The MQTT protocol structures data under topics, which are structured in a topic tree like so:

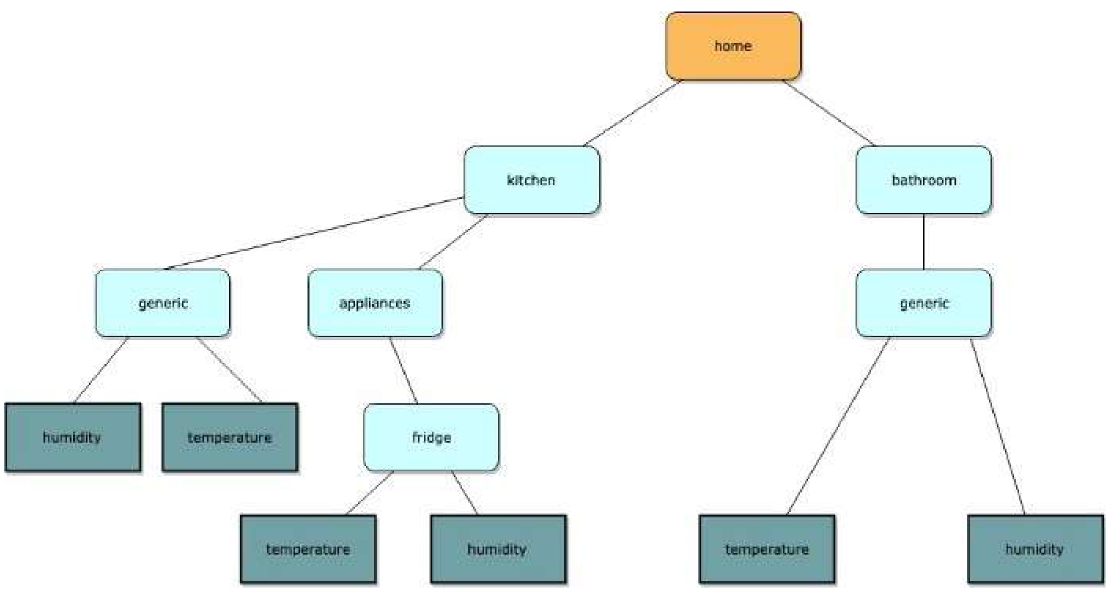

Figure 12 – Topic Tree example (Christos Petropoulos, Hackernoon, 2017)

In this example (Figure 12, Christos Petropoulos, Hackernoon, 2017), if a client wishes to retrieve information on the fridge's temperature, they may subscribe to the “home/kitchen/appliances/fridge/temperature” topic. Topics can contain subtopics (split by the “/” character). It is also possible to subscribe to all of a topic’s subtopics using wildcards: “+” for a single lower-level group of subtopics (for example, subscribing to “home/+” subscribes to both “home/kitchen” and “home/bedroom”, but not to their respective subtopics). If all recursive subtopics are required, the “#” wildcard is used (e.g.: subscribing to “home/kitchen/appliances/#” subscribes to “home/kitchen/appliances/fridge”, “home/kitchen/appliances/fridge/temperature” and “home/kitchen/appliances/fridge/humidity”).

For our Arduino boxes, the following information will be required (Figure 13):

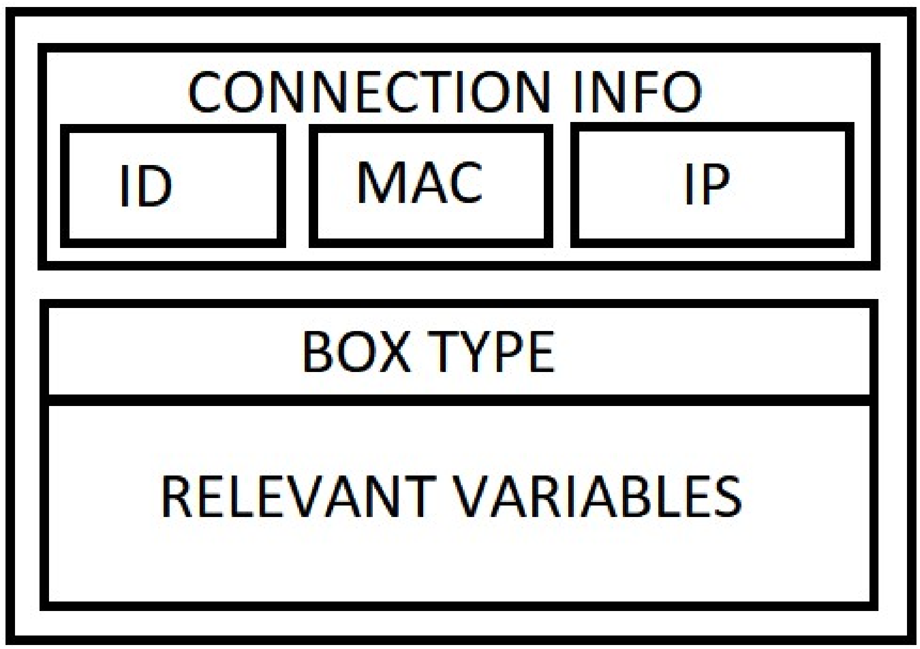

Figure 13 – Box information model

We can then arrange this information on the broker into the following topics (Figure 14):

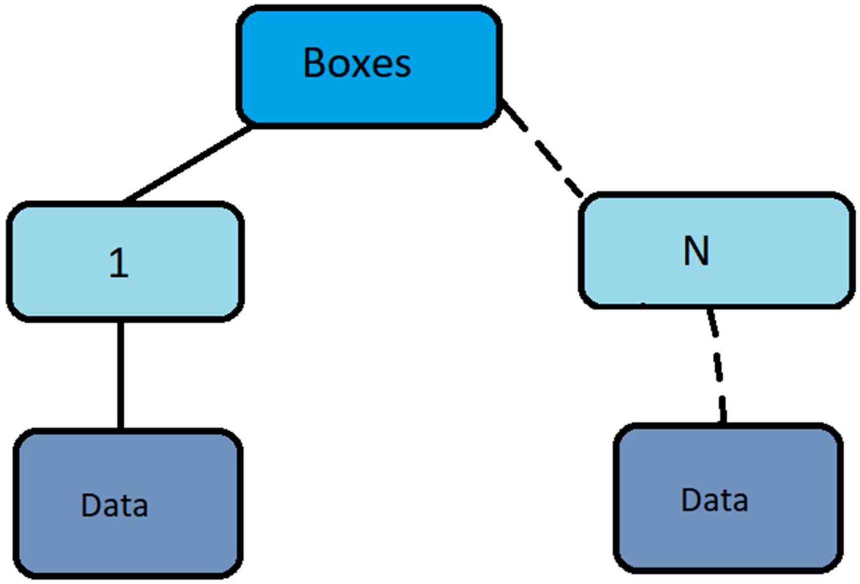

Figure 14 – Broker Topic Tree

The boxes can publish to the topic associated to their ID under the “boxes” topic. With this architecture (Figure 14), they can publish their IP and MAC address values to the broker, which will be published to the subscribed User Interface. The “Data” subtopic will be used by the API to update the box’s state through a JSON object payload, and by the box to periodically publish the information it collects.

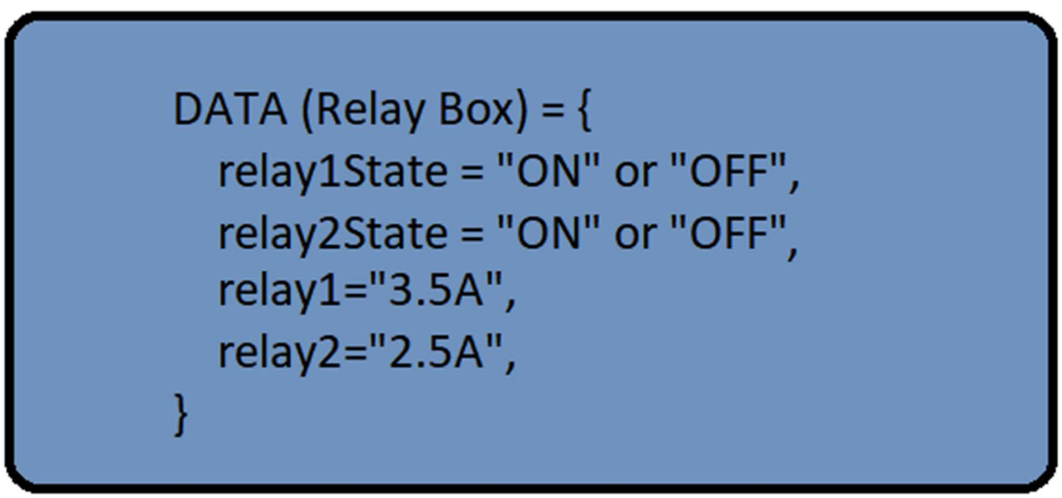

Figure 15 – Relay type topic subtree

In the above (Figure 15) structure, the box will receive onto the “boxes/1/data” topic the new state of its components issued by the React interface through the Express API. It can also publish on this topic to send the new current values it measured (in this example (Figure 15), 3.5 and 2.5 Ampere).
Another box type that was made is one featuring an LCD (Liquid Crystal Display) screen to print out text sent by a user. The screen could display up to four lines with 20 characters each. With the new system, its data subtopic architecture should resemble something like this (Figure 16):

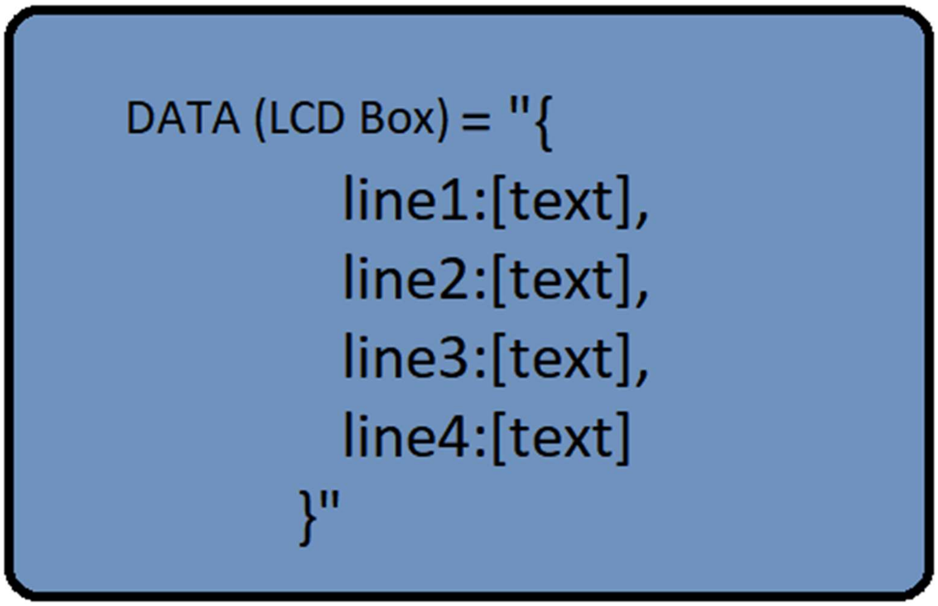

Figure 16 – LCD type subtopic tree

```
#include <SPI.h>
#include <Ethernet.h>
#include <PubSubClient.h>
#inclue <ArduinoJson.h>

byte mac[] = {
  0xDE, 0xED, 0xAA, 0xAA, 0xFE, 0xED
};

EthernetClient ethernetClient;

IPAddress broker(192, 168, 0, 65);
void onMessageReceive(char* topic, byte* payload, unsigned int length);
PubSubClient client(broker, 80, onMessageReceive, ethernetClient);

//The JSON document that will hold the state
StaticJsonDocument<200> state;
StaticJsonDocument<50> data;
//Initial state
char json[] =
"{\"id\":\"1\"\"mac\":\"dead.aaaa.feed\",\"IP\":\"0000.0000.0000\",\"data\":{\"relay1State\":\"ON\",\"relay2State\":\"ON\"}}";
String macString = "dead.aaaa.feed";
String relay1State = "ON";
String relay2State = "ON";
```

Figure 17 – Example MQTT snippet (Part 1)

The preceding snippet (Figure 17) features the initial declarations of an example sketch for an Arduino Uno using Nick O'Leary's PubSubClient library and ArduinoJSON. We initially declare the board's MAC address, the broker's IP address, and the callback function (here, it is named "onMessageReceive"). This callback function will be invoked whenever the MQTT client instance receives a message related to a topic it subscribed to. We then declare the JSON document that will hold the state and the subdocument that will hold the data. Finally, we declare the variables that will be serialised into the state.

```
void setup() {
  Ethernet.begin(mac);
  Serial.begin(9600);
  //Convert the obtained IP into a char array
  IPAddress localAddr = Ethernet.localIP();
  byte byte1 = localAddr[0];
  byte byte2 = localAddr[1];
  byte byte3 = localAddr[2];
  byte byte4 = localAddr[3];
  char IPString[20];
  sprintf(IPString, byte1, byte2, byte3, byte4);
  if(client.connect("Arduino1")){
    String payload = "";
    String dataPayload = "";
    data["relay1State"] = relay1State;
    data["relay2State"] = relay2State;
    serializeJson(data, dataPayload);
    state["IP"] = IPString;
    state["mac"] = macString;
    state["data"] = dataPayload;
    serializeJson(state, payload);
    client.publish("boxes", payload);
    client.subscribe("boxes/1");
  }
}

void loop() {
  client.loop();
}

void onMessageReceive(char* topic, byte* payload, unsigned int length){
  //Print out the contents of the received message
  Serial.println(topic);
  for(int i=0;i<length;i++){
    Serial.print(payload[i]);
  }
  Serial.println();
}
```

Figure 18 - Example MQTT snippet (Part 2)

In the *setup()* procedure, we begin by initialising the Ethernet instance with the declared mac address, and the library emits the required DHCP Discover messages to obtain an IP address and the rest of the network configuration from a DHCP server. We then convert the obtained address from the IPAddress type to a char array. After that, we attempt a connection as ClientID “Arduino1”. We then serialise the necessary variables into the JSON object that will represent the Arduino’s state, and publish it under the “boxes” topic. Finally, we subscribe to the “boxes/1” topic to be notified whenever new commands are issued. It is important to note here that if neither the “boxes” or “boxes/1” topics exist on the broker they are automatically created with the right underlying tree structure by the Mosca broker.

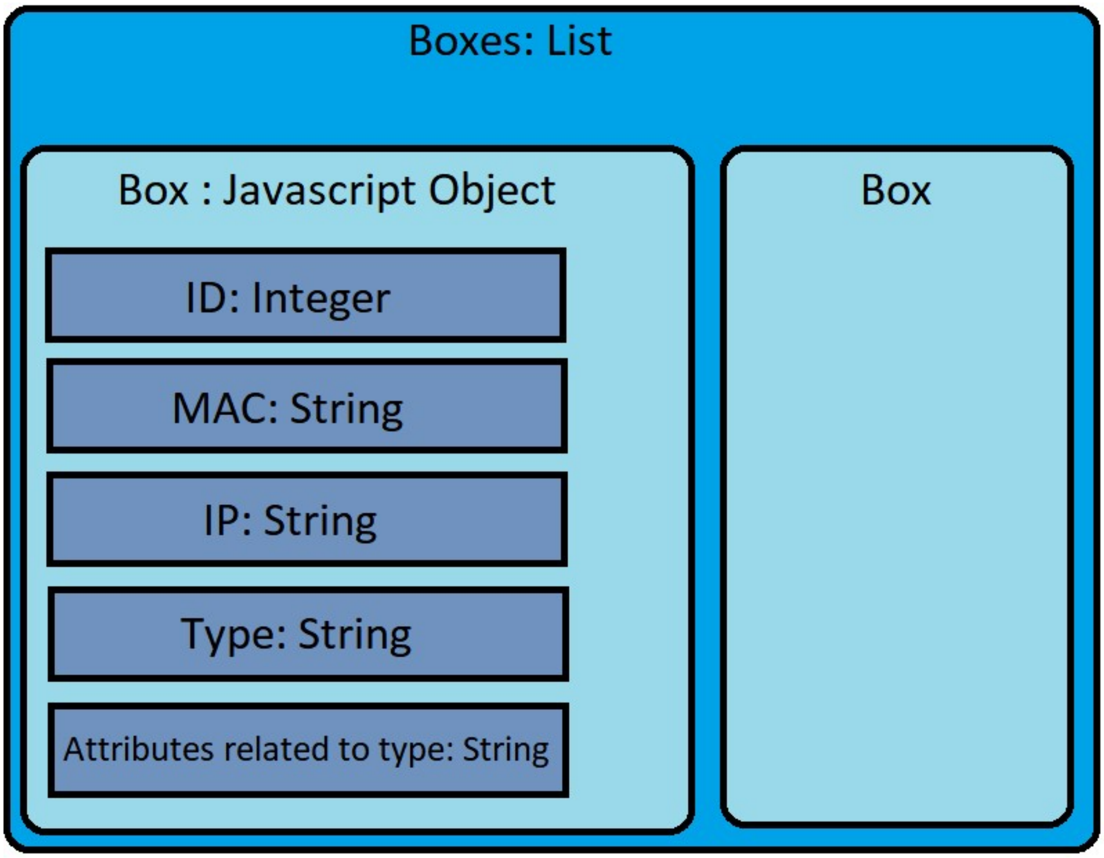

Figure 19 – Express API data model

For the Express API, the data model will consist of a list of items, each mirroring the inner state of their corresponding Arduino. As depicted above (Figure 19), the API will possess a collection of boxes (initially empty), that will get a new item each time a box publishes its information to the broker on start-up. The interface will also be listening for disconnection messages to remove Arduino boxes that go offline from the list.

Each item will contain the box’s ID, its MAC address, its IP address and its type as a string. It will then contain several other strings related to the box’s type. In the case of a relay type, four additional strings containing the state of the relays and the current values measured by each of them.

3.2 Mock-ups

The home screen of the application should remain simple and avoid clutter: only the name of the application and the list of connected devices should be present. In the event that no device has published its information to the broker, a message is shown as follows (Figure 20).

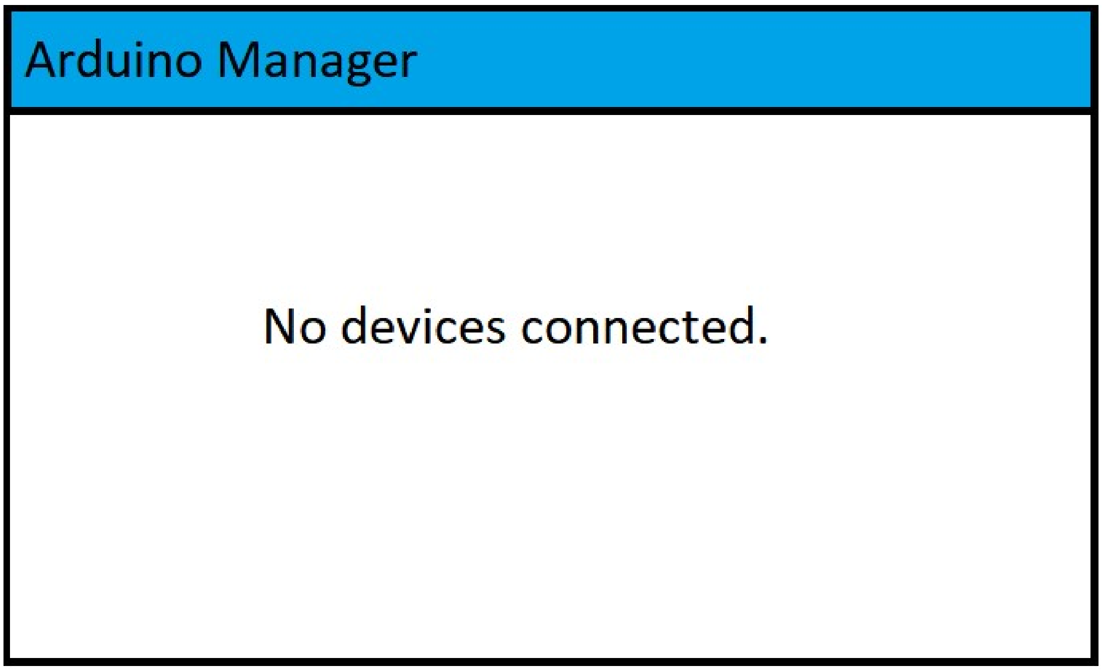

Figure 20 – Home screen mock-up (without connected devices)

Each device should then appear with a widget that makes it explicit they are distinct from each other, and display their corresponding information like so (Figures 21 and 22):

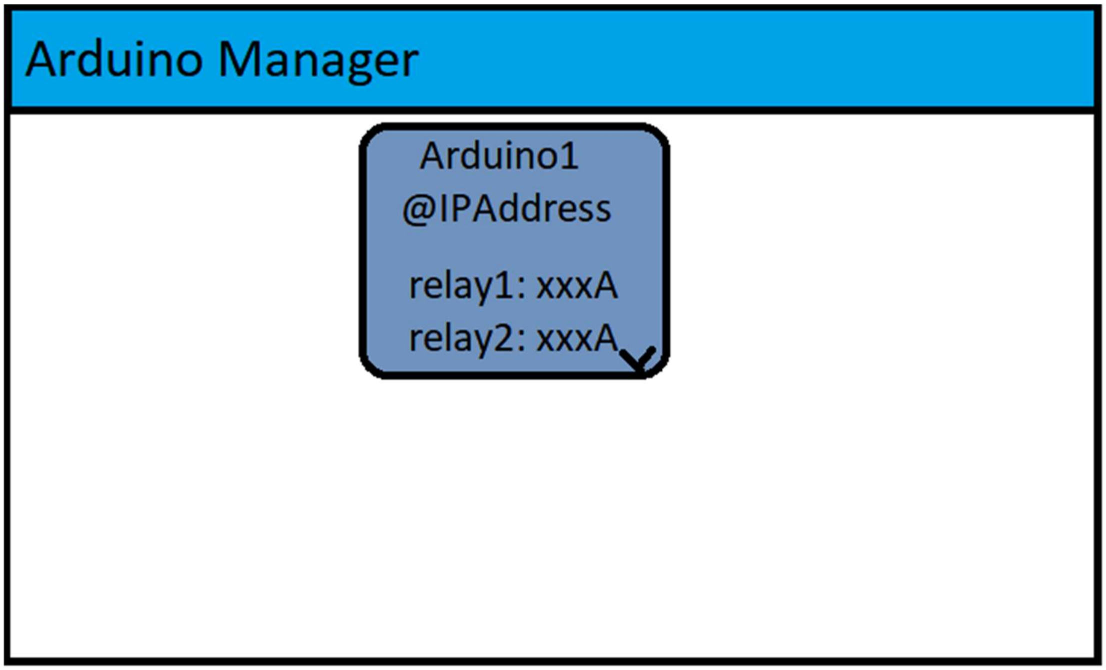

Figure 21 – Home screen mock-up (One Relay device connected)

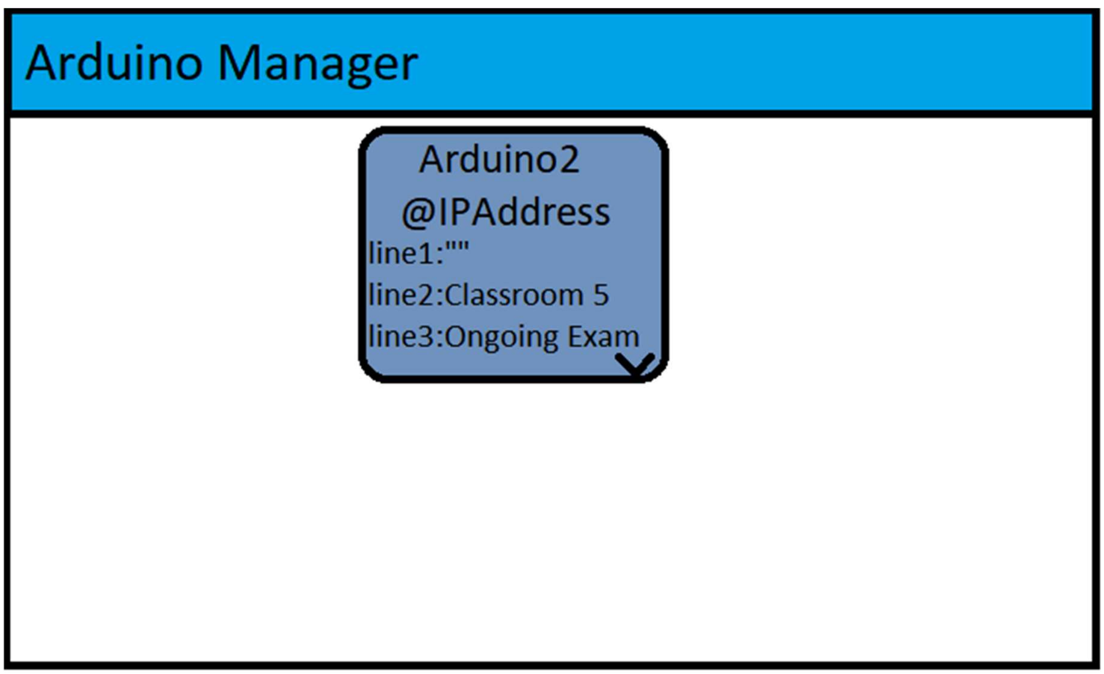

Figure 22 – Home screen mock-up (One LCD device connected)

Upon clicking on the box's "expand" button (bottom right corner of the widget), the widget expands to display a set of controls relevant to the box type (Figures 23 and 24).

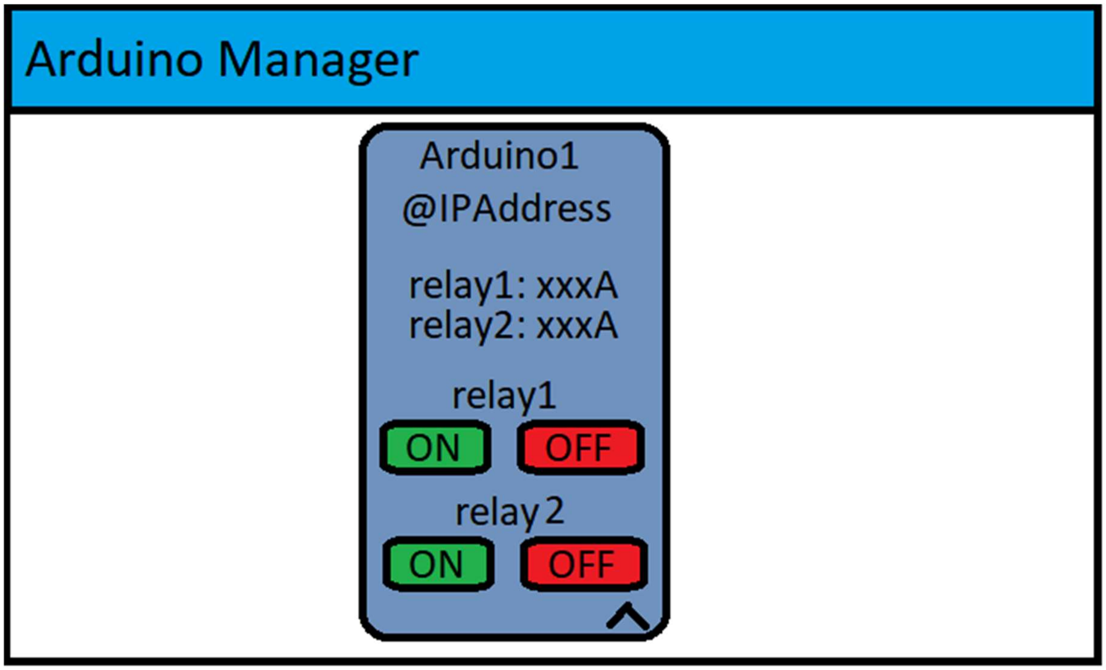

Figure 23 – Relay box control screen mock-up

This screen must display the box's connectivity information (IP and MAC addresses), as well as the current state of its physical components. In this example (Figure 23), both electrical relays can be controlled by clicking on the appropriate button on the interface.

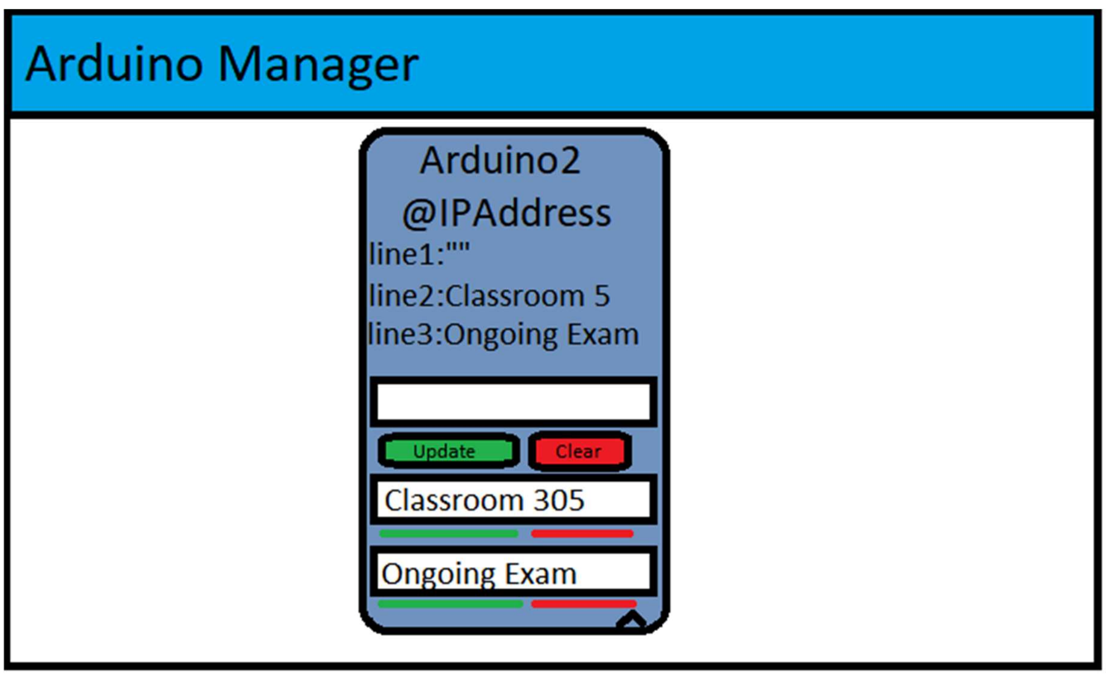

Figure 24 – LCD box control screen mock-up

Display for the LCD boxes (Figure 24) should show each line and have the possibility to clear each individually or all at once.

3.3 Mosca broker implementation

The project’s MQTT broker has been implemented using Mosca, a lightweight JavaScript library developed by Matteo Collina under the MIT license. The first goal was to create a broker module for our Express server.

The following figure (Figure 25) shows a snippet of the Mosca MQTT broker module implementation.

```
/*
 * This module runs the MQTT broker instance and displays messages on the console
 */
module.exports = {
  broker: () => {
    let mosca = require("mosca");
    let moscaSettings = {
      port: 1883,
    };
    let server = new mosca.Server(moscaSettings);
    server.on("ready", () => {
      console.log("Mosca instance initialised and running");
    });
    server.on("clientConnected", (client) => {
      console.log("Client connected with ID ", client.id);
    });
    server.on("published", (packet, client) => {
      console.log(
        `Broker: new message on topic ${packet.topic.toString()}:${packet.payload.toString()}`
      );
    });
  },
```

Figure 25 – Mosca broker implementation

This configuration is the minimum required to run a private MQTT broker, and as such does not comply with best security practices (the likes of authentication or encryption). However, as they are not included in the scope of this project, the matter is inconsequential and default configuration has been maintained for this broker.

```
let app = express();
app.use(cors());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// view engine setup
app.set("views", path.join(__dirname, "views"));
app.set("view engine", "jade");

app.use(logger("dev"));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, "public")));

//Importing and activating the broker module
let MQTT = require("./MQTT.js");
MQTT.broker();

let apiClient = require("./Client.js");
let API = require("./API.js");
let apiInstance = new API(new apiClient());

app.use("/", indexRouter);
apiRouter.get("/boxes", (req, res) => {
  res.json(apiInstance.boxes);
});
apiRouter.put("/boxes", (req, res) => {
  console.log(`API: PUT request:`);
  console.log(req.body);
  apiInstance.updateBox(req.body);
  return res.json(req.body);
});
app.use("/api", apiRouter);
```

Figure 26 – Express server snippet

In the above figure (Figure 26) highlighted in red is where the MQTT broker is invoked in our Express app configuration.

3.4 Express API implementation

To link the React UI to the Mosca broker, an Express API has been implemented. That way, the React interface simply fetches the list of boxes to displays data and commands accordingly and sends new states to the boxes through PUT requests. To minimise the number of apps running on my test environment, the Express API runs on the same Express app as the Mosca broker. To have the API connect to the broker and subscribe to the “boxes/#” topic, an MQTT Client instance had to be implemented and passed to the API module.

```
module.exports = function () {
  this.mqtt = require("mqtt");
  this.client = null;
  this.connect = () => {
    this.client = this.mqtt.connect("http://localhost:1883", {
      clientId: "API",
    });
  };
  this.publish = function (topic, message) {
    this.client.publish(topic, message);
  };
  this.subscribe = function (topic) {
    this.client.subscribe(topic);
  };
  this.onMessage = function (callback) {
    this.client.on("message", callback);
  };
};
```

Figure 27 – Express API MQTT Client implementation

The above snippet (Figure 27) displays the implementation of the MQTT Client that will handle all communication with the broker through publish and subscribe messages.
The API instance is called after the MQTT broker in the Express app script, highlighted in purple previously (Figure 26). Every call to the API, whether they are GET or PUT requests, are handled through the “/api/boxes” endpoint. These calls are then processed using the API instance’s data or methods.

```
module.exports = function (clientInstance) {
  this.boxes = [];
  clientInstance.connect();
  clientInstance.publish("hello", "hello");
  clientInstance.subscribe("boxes/#");
  clientInstance.subscribe("$SYS/#");
  clientInstance.onMessage((topic, message) => {
    console.log(`API: new message on topic ${topic}: ${message}`);
    //Handle new box
    if (topic === "boxes") {
      console.log("New box: parsing data...");
      let box = JSON.parse(message);
      console.log(box);
      this.boxes.push(box);
      clientInstance.publish(`boxes/${box.id}`, `Arduino${box.id} registered`);
    }
    //Handle box removal
    if (topic.includes("disconnect")) {
      let id = message.slice(message.indexOf("Arduino") + 7);
      console.log(`Disconnect: ${id}`);
      this.removeBox(id);
    }
  });
  //Function that returns the box with the specified id
  this.findBoxById = (id) => {
    return this.boxes.find((element) => element.id === id);
  };
  //Function that removes the box with the specified id
  this.removeBox = (id) => {
    let index = this.findBoxById(id);
    this.boxes.splice(index, 1);
  };
  //Function that replaces a box with updated information
  this.updateBox = (newBox) => {
    let index = this.findBoxById(newBox.id);
    this.boxes.splice(index, 1, newBox);
    clientInstance.publish(`boxes/${newBox.id}`, `${JSON.stringify(newBox)}`);
  };
};
```

Figure 28 – Express API module implementation

The API instance requires an instance of the MQTT Client to function, as its first operation is connecting to the broker and subscribing to the “boxes/#” topic to be notified when new boxes are connected, and subscribing to the “$SYS” topic to be notified when they disconnect.

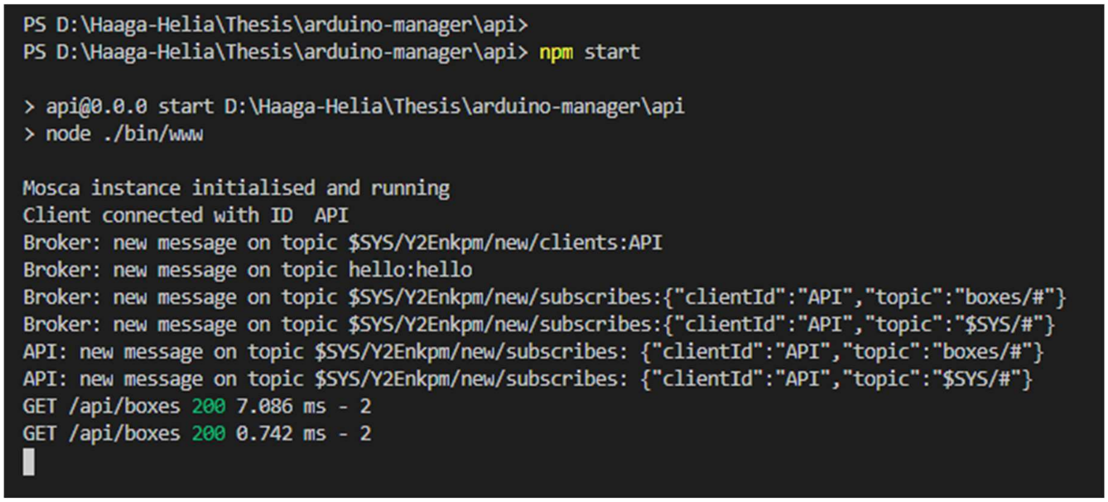

Figure 29 – Express server runtime log

In the above figure (Figure 29), we have started the Express Server and queried the list of boxes using Postman (Figure 30).

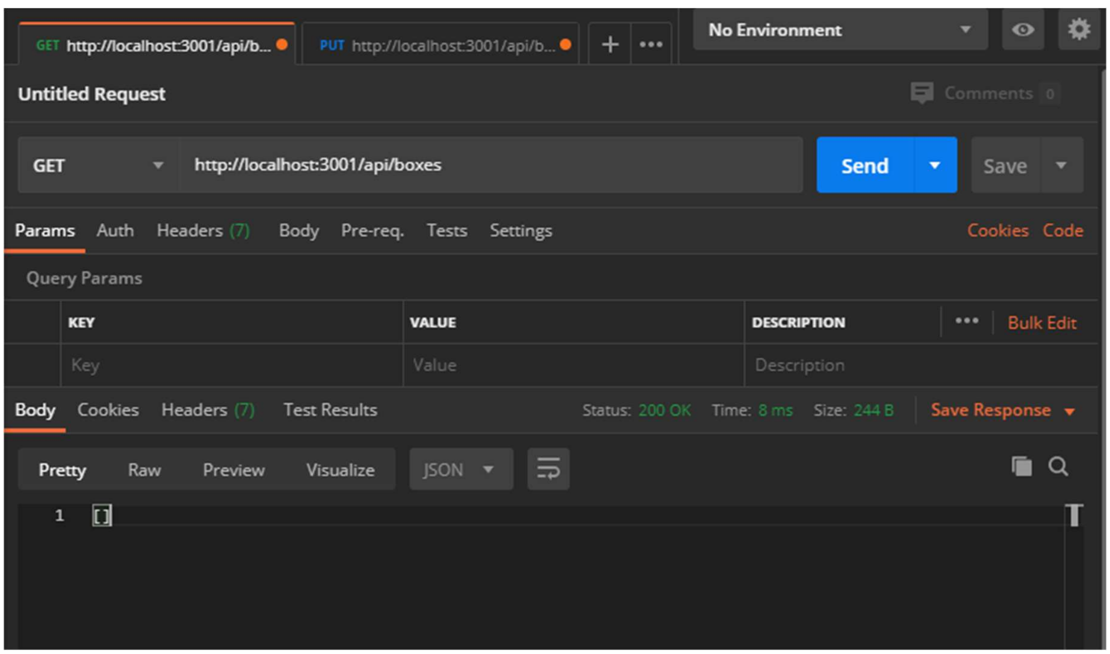

Figure 30 – Postman test 1: Fetching the initial list of boxes

The request returned an HTTP response code 200, meaning everything went as expected. We have successfully recovered the list of boxes, which is presently empty as no device has published anything yet.

3.5 Test JavaScript Publisher

I was unable to obtain an Arduino board with an Ethernet module to use for testing purposes due to various coronavirus-related circumstances. Therefore, testing of publishing was done with a JavaScript publisher app that would simulate messages from an Arduino box.
This mock publisher was implemented on a different Express app and features two MQTT clients: one representing a relay-type box, and the other representing an LCD-type box.

```
let mqtt = require("mqtt");
let client;

module.exports = {
  init: () => {
    console.log("RelayPublisher startup succesful");
  },
  connect: () => {
    console.log("Connecting to broker...");
    client = mqtt.connect("http://localhost:1883", { clientId: "Arduino1" });
    client.on("message", (topic, message) => {
      console.log(`Arduino1: new message on topic ${topic}: ${message}`);
    });
  },
  publish: () => {
    console.log("Publishing INFO...");
    let data = {
      id: 1,
      IP: "192.168.0.91",
      mac: "dead.aaaa.feed",
      type: "Relay",
      data: {
        relay1Status: "ON",
        relay2Status: "ON",
        relay1: "3.5A",
        relay2: "2.5A",
      },
    };
    client.publish("boxes", JSON.stringify(data));
  },
  subscribe: () => {
    console.log("Subscribing to self topic...");
    client.subscribe("boxes/1");
  },
};
```

Figure 31 – Mock relay box snippet

The above snippet (Figure 31) features the mock relay box module’s functions that emulate the behaviour of a box containing relays connecting to the broker.

```
let mqtt = require("mqtt");
let client;

module.exports = {
  init: () => {
    console.log("LCDPublisher startup succesful");
  },
  connect: () => {
    console.log("Connecting to broker...");
    client = mqtt.connect("http://localhost:1883", { clientId: "Arduino2" });
    client.on("message", (topic, message) => {
      console.log(`Arduino2: new message on topic ${topic}: ${message}`);
    });
  },
  publish: () => {
    console.log("Publishing INFO...");
    let data = {
      id: 2,
      IP: "192.168.0.92",
      mac: "dead.bbbb.feed",
      type: "LCD",
      data: {
        line1: "",
        line2: "Classroom 305",
        line3: "Ongoing Exam",
        line4: "",
      },
    };
    client.publish("boxes", JSON.stringify(data));
  },
  subscribe: () => {
    console.log("Subscribing to self topic...");
    client.subscribe("boxes/2");
  },
};
```

Figure 32 – Mock LCD box snippet

The above snippet (Figure 32) features the mock LCD box module’s functions that emulate the behaviour of a box with an LCD screen mounted on it connecting to the broker.

```
//Starting the publishers
let RelayPublisher = require("./script/RelayPublisher.js");
RelayPublisher.init();
RelayPublisher.connect();
RelayPublisher.publish();
RelayPublisher.subscribe();

let LCDPublisher = require("./script/LCDPublisher.js");
LCDPublisher.init();
LCDPublisher.connect();
LCDPublisher.publish();
LCDPublisher.subscribe();
```

Figure 33 – Express MQTT Publisher app snippet

In the above visual (Figure 33), we invoke each publisher's methods in succession to simulate a new Arduino device connecting to the MQTT broker. They then subscribe to their respective topics to receive messages related to their commands. We can now start testing with Postman to confirm that the collection of boxes is properly created on the Express API app.

The first step is visually confirming that data has been received by the appropriate modules on the Express API app:

```
Client connected with ID  Arduino1
Broker: new message on topic $SYS/CIxgZLL/new/clients:Arduino1
Client connected with ID  Arduino2
API: new message on topic $SYS/CIxgZLL/new/clients: Arduino1
API: new message on topic $SYS/CIxgZLL/new/clients: Arduino2
Broker: new message on topic $SYS/CIxgZLL/new/clients:Arduino2
Broker: new message on topic boxes:{"id":1,"IP":"192.168.0.91","mac":"dead.aaaa.feed","type":"Relay
","data":{"relay1Status":"ON","relay2Status":"ON","relay1":"3.5A","relay2":"2.5A"}}
Broker: new message on topic boxes:{"id":2,"IP":"192.168.0.92","mac":"dead.bbbb.feed","type":"LCD",
"data":{"line1":"","line2":"Classroom 305","line3":"Ongoing Exam","line4":""}}
API: new message on topic boxes: {"id":1,"IP":"192.168.0.91","mac":"dead.aaaa.feed","type":"Relay",
"data":{"relay1Status":"ON","relay2Status":"ON","relay1":"3.5A","relay2":"2.5A"}}
New box: parsing data...
{ id: 1,
  IP: '192.168.0.91',
  mac: 'dead.aaaa.feed',
  type: 'Relay',
  data:
   { relay1Status: 'ON',
     relay2Status: 'ON',
     relay1: '3.5A',
     relay2: '2.5A' } }
API: new message on topic boxes: {"id":2,"IP":"192.168.0.92","mac":"dead.bbbb.feed","type":"LCD","d
ata":{"line1":"","line2":"Classroom 305","line3":"Ongoing Exam","line4":""}}
New box: parsing data...
{ id: 2,
  IP: '192.168.0.92',
  mac: 'dead.bbbb.feed',
  type: 'LCD',
  data:
   { line1: '',
     line2: 'Classroom 305',
     line3: 'Ongoing Exam',
     line4: '' } }
API: new message on topic $SYS/CIxgZLL/new/subscribes: {"clientId":"Arduino1","topic":"boxes/1"}
API: new message on topic $SYS/CIxgZLL/new/subscribes: {"clientId":"Arduino2","topic":"boxes/2"}
Broker: new message on topic $SYS/CIxgZLL/new/subscribes:{"clientId":"Arduino1","topic":"boxes/1"}
Broker: new message on topic $SYS/CIxgZLL/new/subscribes:{"clientId":"Arduino2","topic":"boxes/2"}
Broker: new message on topic boxes/1:Arduino1 registered
Broker: new message on topic boxes/2:Arduino2 registered
API: new message on topic boxes/1: Arduino1 registered
API: new message on topic boxes/2: Arduino2 registered
```

Figure 34 – Express server runtime log: new clients connected

As we can see from the log (Figure 34), there is quite a lot going on behind the scenes. What we first note is that both publisher clients have successfully connected to the broker, and they have published their current state. Then, we can see that the API has issued a publish message to both the “boxes/1” and “boxes/2” topics to notify the boxes they have been registered.

```
PS D:\Haaga-Helia\Thesis\arduino-manager\test-publisher>
PS D:\Haaga-Helia\Thesis\arduino-manager\test-publisher> npm start

> test-publisher@0.0.0 start D:\Haaga-Helia\Thesis\arduino-manager\test-publisher
> node ./bin/www

RelayPublisher startup succesful
Arduino1: Connecting to broker...
Arduino1: Publishing INFO...
Arduino1: Subscribing to self topic...
LCDPublisher startup succesful
Arduino2: Connecting to broker...
Arduino2: Publishing INFO...
Arduino2: Subscribing to self topic...
Arduino1: new message on topic boxes/1: Arduino1 registered
Arduino2: new message on topic boxes/2: Arduino2 registered
```

Figure 35 – Express Publisher runtime log

Since the mock boxes have subscribed to their respective topic, they are properly notified that they have been registered by the API as shown in the figure above (Figure 35). It is worth noting that the logs are not issued by the main Express app, but by the modules themselves, which helps us confirm that they are receiving the right messages from the right topics.

Finally, we can confirm that the API has properly updated its list of boxes by making a GET request to its “/api/boxes” endpoint using postman like so (Figure 36):

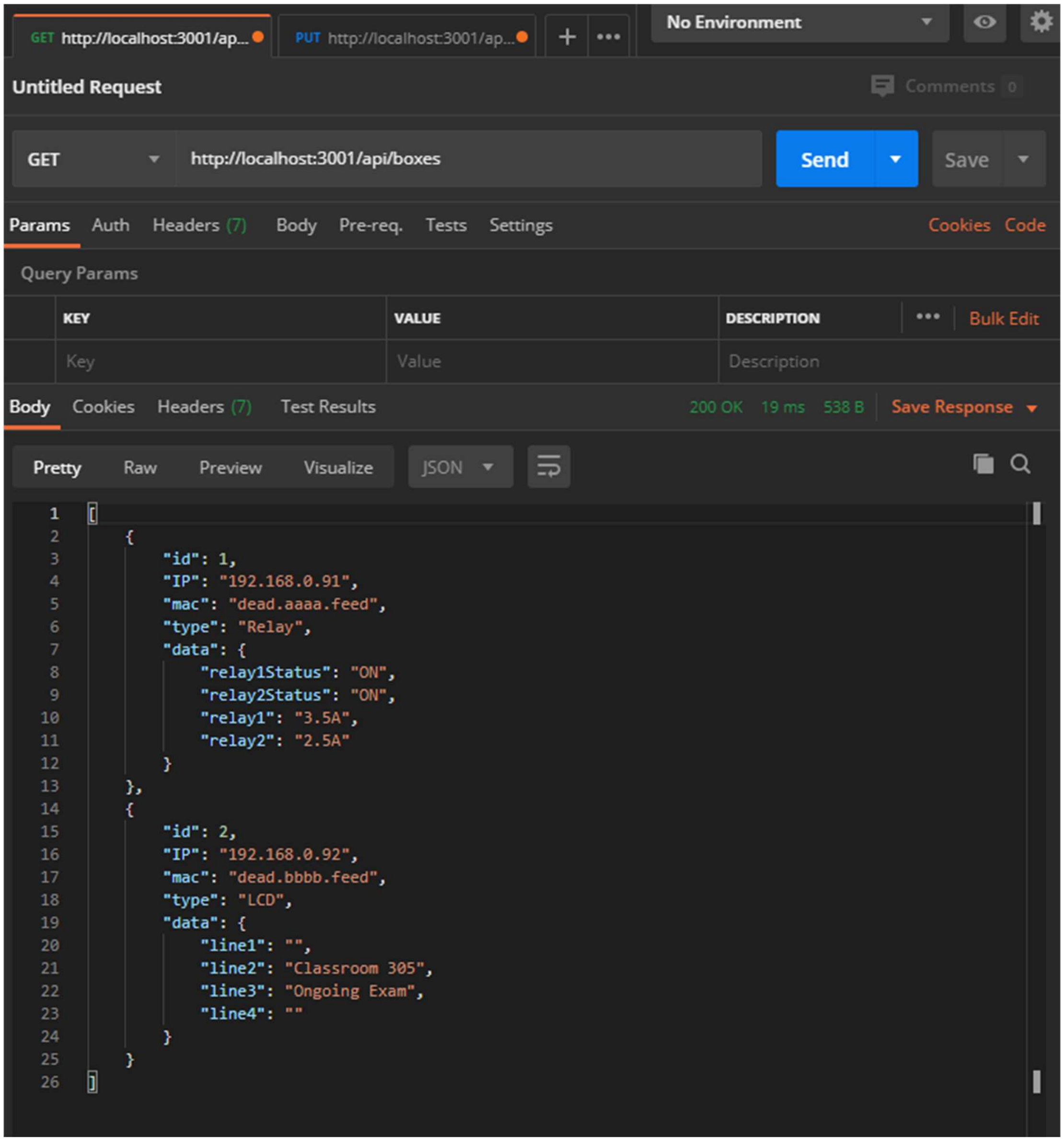

Figure 36 – Postman test 2: fetching the newly published boxes

This structure will be what the React interface uses when editing the state of the boxes through PUT requests to the API. It will also be the core behind displaying the information of each box, as the “type” attribute will help determine what components to render for each box.

3.6 React UI implementation

The majority of the interface design has been implemented using the open-source Material-ui library. It provides a wide array of elegant components following Google’s Material Design guidelines for user experience. (Material-ui, 2020)

The reasoning behind this choice is that Material Design is "*an adaptable system of guidelines, components, and tools that support the best practices of user interface design*" (Kaylin Berg, 2019). It is relatively safe to assume that Google would be quite knowledgeable on the matter as their spending in research and development towards collecting the best design techniques and styles had reached 2.4 billion dollars at the time the article was written. (Kaylin Berg, 2019)

Here is what the initial home screen looks like (Figure 37):

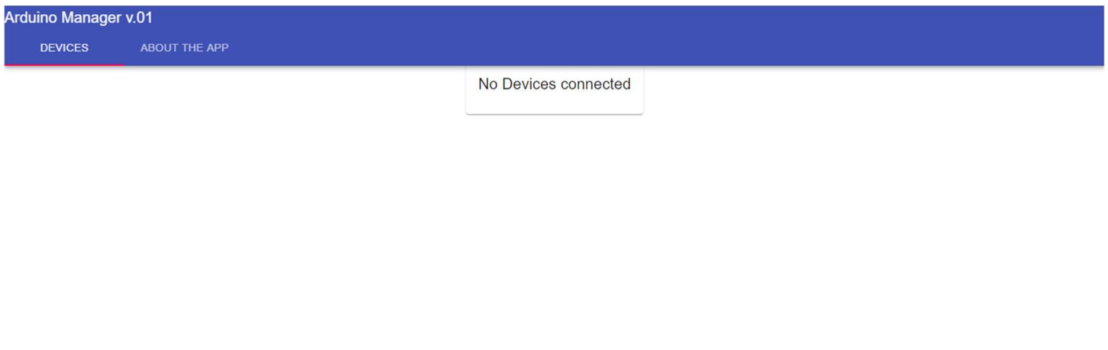

Figure 37 – App Screenshot: home screen

Initially, no device has published anything to the broker, therefore the UI has no boxes to display. However, when we start up the mock publishers, new devices appear on screen (Figure 38):

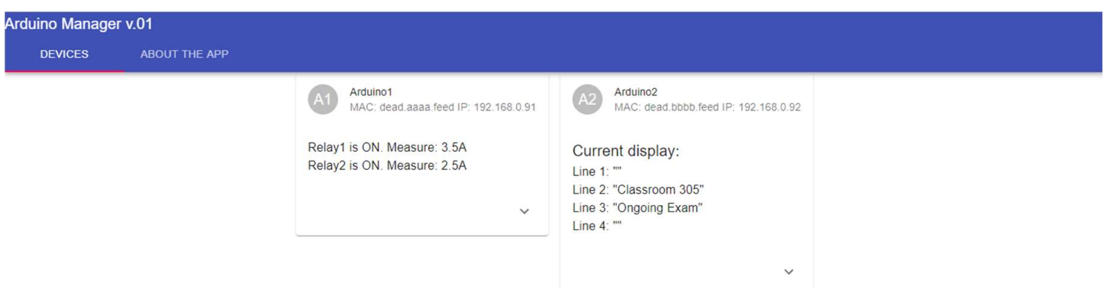

Figure 38 – App Screenshot: new devices connected

The "Card" Material-ui component was ultimately chosen as the container for displaying a box's information. It possesses intuitive features such as expanding and collapsing. Which is how we access a box's commands: by clicking on the bottom-right arrow on a card, we can expand the card to display commands associated to the box type like so (Figure 39):

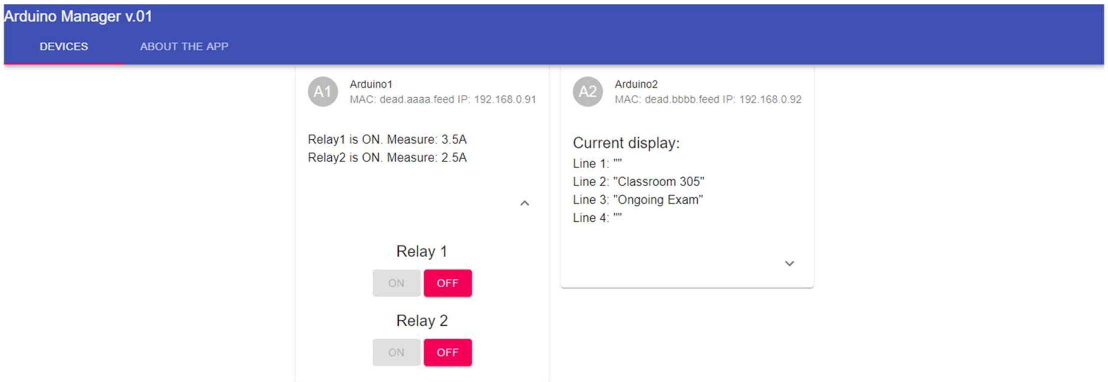

Figure 39 – App Screenshot: Relay box commands

Arduino1 is a relay-type box, therefore the app renders “Button” Material-ui components to control their state. Each click on them fires a PUT request to the API and changes their state as follows (Figure 40):

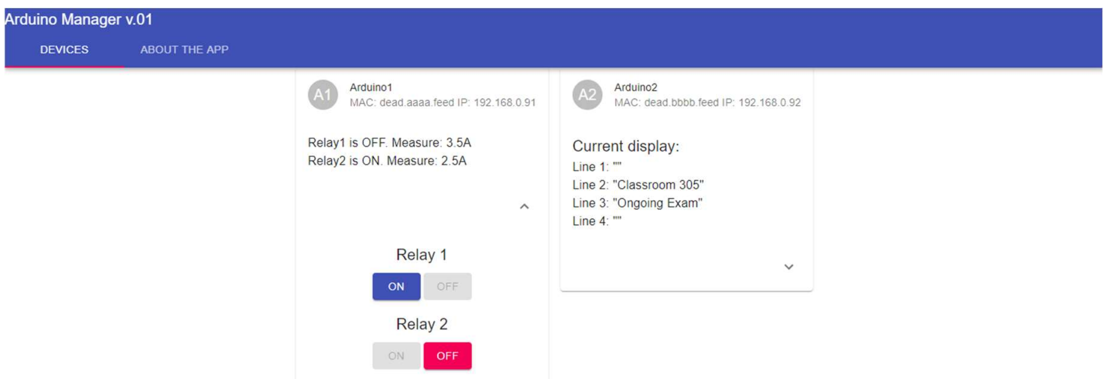

Figure 40 – App Screenshot: Changing a relay’s state

The state is immediately changed, and now Relay1 is switched off. It is worth noting that it is only possible to change the state of the relay to a state it is not in, hence why the other button is now greyed out and disabled. We can confirm on the mock Arduino’s logs that they have indeed received the appropriate publish message (Figure 41):

```
Arduino1: new message on topic boxes/1: {"id":1,"IP":"192.168.0.91","mac":"dead.aaaa.feed","type":"
Relay","data":{"relay1Status":"OFF","relay2Status":"ON","relay1":"3.5A","relay2":"2.5A"}}
```

Figure 41 – Mock Arduino logs: state change

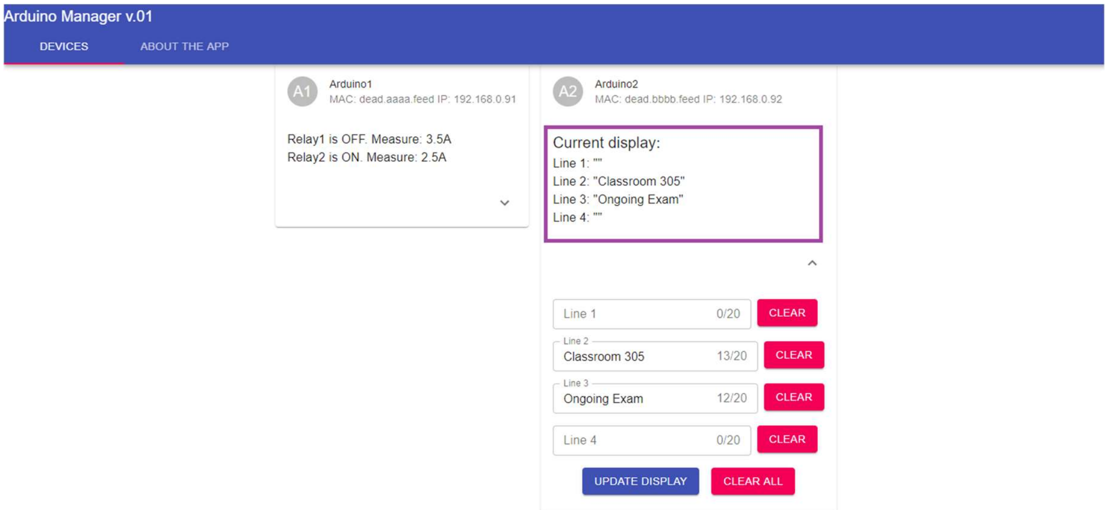

Figure 42 – App Screenshot: LCD box commands

For the LCD box, one of the goals was that the display would be adaptable to how many lines the screen possessed. Therefore, display and command interfaces were designed with adaptability in mind.

```
import React from "react";
import { Grid, Typography } from "@material-ui/core";

const LCDBoxDataDisplay = (props) => {
  const box = props.box;
  return (
    <Grid container direction="column">
      <Grid item>
        <Typography variant="h6">Current display:</Typography>
      </Grid>
      <Grid item>
        {box.data.map((line, index) => (
          <Typography key={index} variant="body1">
            Line {index + 1}: "{line}"
          </Typography>
        ))}
      </Grid>
    </Grid>
  );
};

export default LCDBoxDataDisplay;
```

Figure 43 – LCD Data display snippet

The above snippet (Figure 43) shows how the data display section of the card (highlighted in purple in Figure 42) was implemented for the LCD box card. For each line of data the box possesses, a Typography component is generated. The same principle was used when implementing the TextField components for data editing. As that snippet is quite long, it will not be shown in a screenshot, but the code is available on GitHub (see Appendix 1: React interface).

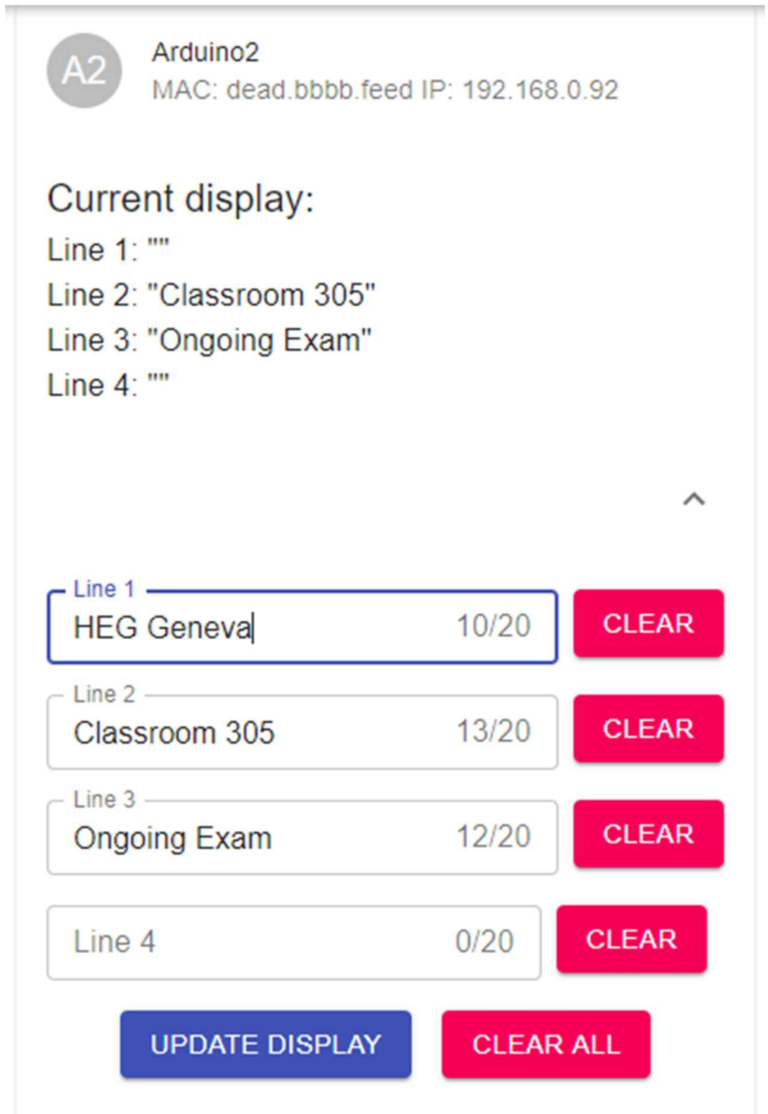

Figure 44 – App Screenshot: editing lines part one

As shown in the above visual (Figure 44), it is possible to edit text in the command lines, and the current character count is displayed next to the input's maximum size. However, the data will not be sent until the "Update Display" button is clicked. The reasoning behind this choice is that opting for sending data on every change of the textbox would trigger a call to the API and a publish message to the Arduino every time a character would be entered or deleted. To remain within the objective of preserving the Arduino's computing resources, the more lightweight approach was chosen.

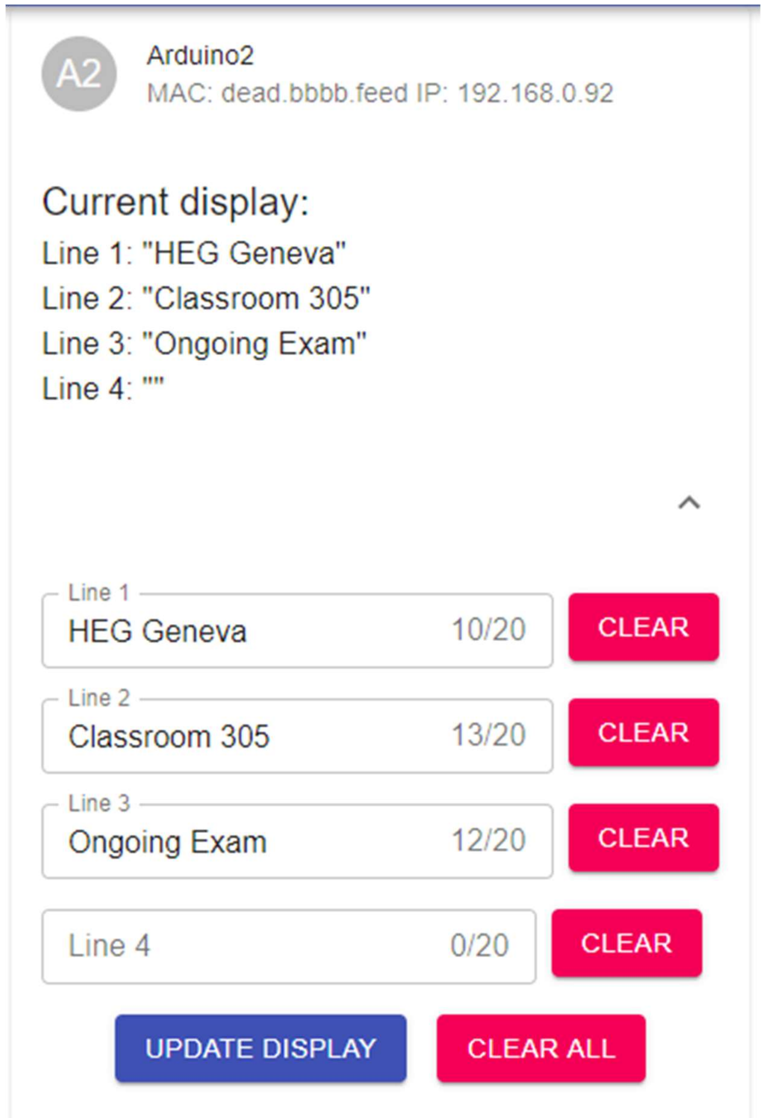

Figure 45 – App Screenshot: editing lines part two

From the preceding visual (Figure 45), we can observe that data has been changed in the data display. Since it is retrieved from the HomeScreen component, which retrieves it from the API directly, we can conclude that data has been modified properly.

Clicking on any of the "Clear" buttons will clear the line, but again, data will not be modified unless the "Update Display" button is clicked. The following figure (Figure 46) demonstrates the state of the LCD card after the "Clear all" and "Update Display" buttons have been pressed in sequence:

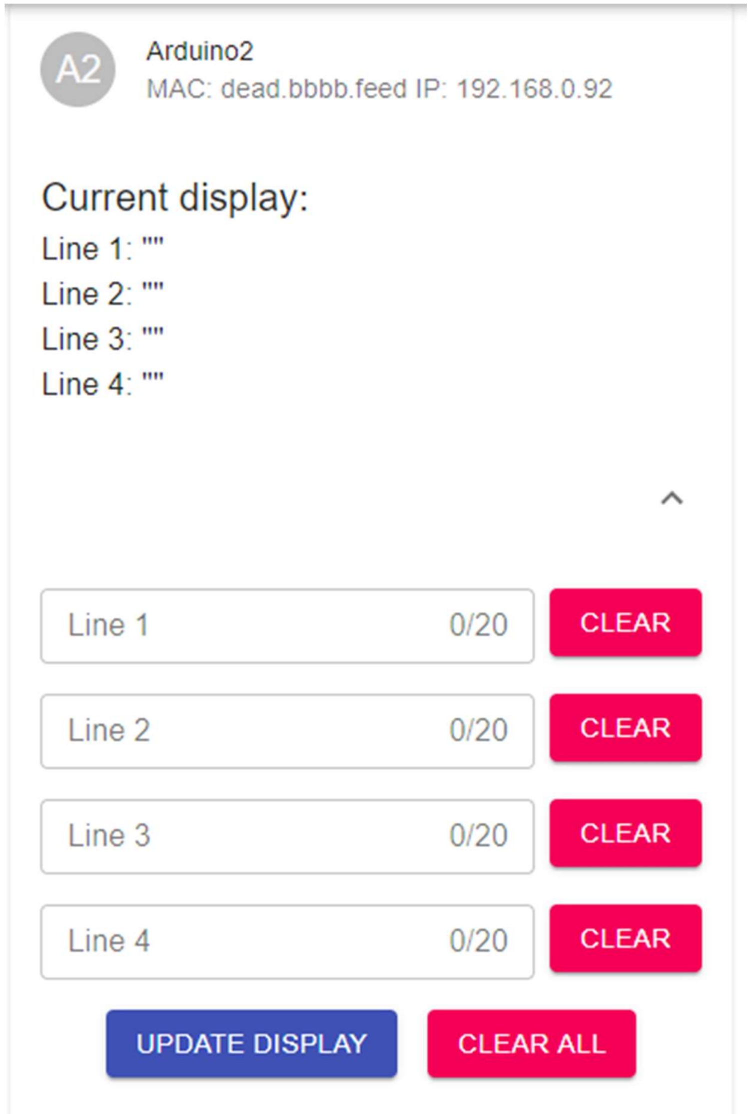

Figure 46 – App Screenshot: clearing lines

In the current state of the app, it is not possible to collapse the commands, or navigate to the “About the App” tab (shown in the top-left corner of Figures 38, 39, 40 and 42) without losing the inputted data. To do so would require implementation of a Redux store to persist the entire application’s state and falls outside of the scope of this project.

4 – Discussion and Conclusion

This project was inspired by my internship at HEG Genève. Its objective was to bring a simple and intuitive user interface to manage and control several Arduino devices connected on the network. As one of the main end users for these devices, this objective has been fulfilled. The interface follows Material Design principles thanks to the use of Material-ui components arranged in an orderly fashion. While the colour palette follows the standard React blue theme in the scope of this thesis, the final version follows HEG's colour theme (black, white and red).

I had experience with Material-ui prior to this project, but over its course I have grown fond of ternary operators and their power: they allow insertion of "IF/ELSE" statements with return values in any given context and prove especially useful when rendering components that must adapt displayed elements according to given properties. The most notable topic on which I have learned new things is MQTT. I had previously heard of this protocol, but never looked into it in detail or experimented with it. I had not fully realised its flexibility for communication since from a technical standpoint, communicating with Strings automatically means that it is possible to send JSON objects through this medium. The concept of a topic tree also offers great flexibility to the developer establishing their data structure, since all message-handling features are custom. But this might also be its weakness, since the lack of a default structure can be an obstacle to anyone looking for a simple solution, and establishing a topic tree structure might often require too much work compared to the end result of serving a tech hobby for example.

From a developer standpoint, this project has not been easy: the coronavirus outbreak has massively impacted my workflow with added stress and anxiety. However, now that it is complete, I hope this document can serve as a basis for anyone attempting to create their own management interface, as I will be keeping the repositories related to this project public.

References

Arduino. (24/04/2020). *What is Arduino?* Retrieved from Arduino: https://www.arduino.cc/en/Guide/Introduction

Arduino. (24/04/2020). *Tech Specs.* Retrieved from Arduino: https://store.arduino.cc/arduino-uno-rev3

Arduino (25/04/2020). *Web Server.* Retrieved from Arduino: https://www.arduino.cc/en/Tutorial/WebServer

Arduino (27/04/2020) *EthernetClient()*. Retrieved from Arduino: https://www.arduino.cc/en/Reference/EthernetClient

ArduinoJSON (16/05/2020) *ArduinoJSON*. Retrieved from ArduinoJSON: https://arduinojson.org/

MikeGrusin (14/01/2013). *Serial Peripheral Interface(SPI).* Retrieved from Sparkfun on 30/04/2020: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all

OPC (01/05/2020) *What is MQTT?* Retrieved from OPC Router: https://www.opc-router.com/what-is-mqtt/

Michael Yuan (12/05/2017) *Getting to know MQTT.* Retrieved from IBM on 01/05/2020: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/

Christos Petropoulos (07/08/2019) *A detailed guide to the world of MQTT.* Retrieved from Hackernoon on 01/05/2020: https://hackernoon.com/a-detailed-guide-to-the-world-of-mqtt-bo1d63cay

Nick O'Leary (01/05/2020) *Arduino Client for MQTT.* Retrieved from Knolleary: https://pubsubclient.knolleary.net/

Alif Abdullah (10/01/2018) *Setting up private MQTT broker using Mosca in Node.js.* Retrieved from Medium on 01/05/2020: https://medium.com/@alifabdullah/setting-up-private-mqtt-broker-using-mosca-in-node-js-c61a3c74f952

Marc Höffl (11/07/2019) *mqtt-react.* Retrieved from GitHub on 01/05/2020: https://github.com/KeKs0r/mqtt-react

Matteo Collina (2013) *Mosca.* Retrieved from Mosca on 03/05/2020: http://www.mosca.io/

Material-ui (2020) Material-ui. Retrieved from Material-ui on 20/05/2020: https://material-ui.com/

Vangie Beal (2020) *RAM - random access memory*. Retrieved from Webopedia on 16/05/2020: https://www.webopedia.com/TERM/R/RAM.html

Vangie Beal (2020) *CPU – Central Processing Unit*. Retrieved from Webopedia on 16/05/2020:

https://www.webopedia.com/TERM/C/CPU.html

Vangie Beal (2020) *DHCP – Dynamic Host Configuration Protocol*. Retrieved from Webopedia on 16/05/2020:

https://www.webopedia.com/TERM/D/DHCP.html

RapidTables (2020) *Ampere Unit*. Retrieved from RapidTables on 16/05/2020:

https://www.rapidtables.com/electric/ampere.html

Kaylin Berg (24/05/2019) *What is Google Material Design & how does it affect modern application development?* Retrieved from Illinois Technology Association on 20/05/2020:

https://www.illinoistech.org/news/453051/What-is-Google-Material-Design--How-Does-it-Affect-Modern-Application-Development.htm

JSON (2020) *Introducing JSON*. Retrieved from JSON on 22/05/2020:

https://www.json.org/json-en.html

Mahesh Parahar (26/11/2019) *Difference between MAC Address and IP Address*. Retrieved from TutorialsPoint on 22/05/2020:

https://www.tutorialspoint.com/difference-between-mac-address-and-ip-address

Shana Pearlman (07/09/2016) *What are APIs and how do APIs work?* Retrieved from MuleSoft on 22/05/2020:

https://blogs.mulesoft.com/biz/tech-ramblings-biz/what-are-apis-how-do-apis-work/

Appendices

Appendix 1. GitHub Repositories Links

Express API and MQTT broker: https://github.com/loicprodhom/arduino-manager-api
React Interface: https://github.com/loicprodhom/arduino-manager-client
Test MQTT Publisher: https://github.com/loicprodhom/arduino-manager-test-publisher